PATENT
Attorney Docket No. DE920000098US1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of	
Jens LEENSTRA et al.	Examiner: LI, Aimee J.
Serial No: 09/683,383	Art Unit: 2183
Filed: December 20, 2001	
For: METHOD AND SYSTEM FOR PIPELINE REDUCTION	

APPEAL BRIEF

Board of Patent Appeals and Interferences
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

The applicant submits this brief pursuant to 37 C.F.R. §41.37(a)(1) in furtherance of the Notice of Appeal filed August 16, 2006.

Please charge Deposit Account 50-0510 the $500 fee for filing this Appeal Brief. No other fee is believed due with this Appeal Brief, however, should another fee be required please charge Deposit Account 50-0510.

Real Party In Interest

The real party in interest is International Business Machines Corporation.

Related Appeals And Interferences

None.

Status of Claims

Claims 1-16 are pending in the present Application, with claims 1, 6 and 10 being independent claims. The rejection of claims 1-16 is appealed.

Status of Amendments

No amendments to the claims were made after the Final Office Action dated May 16, 2006 ("FOA").

Summary of the Claimed Subject Matter

The pending application relates to an improved method and system for operating a high frequency out-of-order processor with increased pipeline length. App., [0001]. The application teaches a new scheme to bypass an instruction pipeline by the detection and exploitation of a "no-dependencies" condition. App., [0039]. The "no-dependencies" condition indicates that all required source data is available for the instruction and that one or more stages of the pipeline can be advantageously bypassed. App., [0039].

Independent claim 1 recites one embodiment of the invention as a method for operating an out-of-order processor comprised of an instruction pipeline. App., [0001] and Fig. 2. The method includes, for detection of a dependency, a step of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of the current instruction. App., [0049] and Fig. 7, item 730. The instructions are stored in a temporary buffer associated with a pipeline process downstream of the current instruction. App., [0048] and Fig. 4. A generating step generates a no-dependency signal associated with the current instruction. App., [0062] and Fig. 7, item 770. A bypassing step bypasses a portion of the instruction pipeline for the current instruction if the no-dependency signal is active. App., [0037], [0076] and Fig. 13. An addressing step addresses a mapping-table-entry of a mapping table with a logical source register address of

the current instruction thus determining the mapped physical target register address. App., [0069] and Fig. 9, item 940.

Independent claim 6 recites another embodiment of the invention as a processing system having means for executing a readable machine language. App., [0021]. In the system, a first computer readable code, for the detection of a dependency, determines for each current instruction involved in a renaming process that a logic target address of one or more instructions stored in a temporary buffer associated with a pipeline process downstream of the current instruction is not the same as a logic source address of the current instruction. App., [0048], [0049] and Fig. 4 and Fig. 7, items 730. A second computer readable code generates a no-dependency signal associated with the current instruction. App., [0062] and Fig. 7, item 770. A third computer readable code bypasses a portion of the instruction pipeline for the current instruction if the no-dependency signal is active. App., [0037], [0076] and Fig. 13. A fourth computer readable code addresses a mapping-table-entry of a mapping table with a logical source register address of the current instruction, thus determining the mapped physical target register address. App., [0069] and Fig. 9, item 940.

Independent claim 10 recites a further embodiment of the invention as a computer system having an out-of-order processing system. App., [0021]. In the system, a first computer readable code, for the detection of a dependency, determines for each current instruction involved in a renaming process that a logic target address of one or more instructions stored in a temporary buffer associated with a pipeline process downstream of the current instruction is not the same as a logic source address of the current instruction. App., [0048], [0049] and Fig. 4 and Fig. 7, items 730. A second computer readable code generates a no-dependency signal associated with the current instruction. App., [0062] and Fig. 7, item 770. A third computer readable code assigns an entry in the temporary buffer to the logic source address of said current instruction if the no-dependency signal is

not active. App., [0048]. A fourth computer readable code issues the instruction operand data to an instruction execution unit without assigning the entry in the temporary buffer to the logic source address of said current instruction if the no-dependency signal is active. App., [0037] and Fig. 13. A fifth computer readable code addresses a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address. App., [0069] and Fig. 9, item 940.

Grounds for Rejection to be Reviewed on Appeal

I. Claim 10 is indicated as substantially duplicative to claim 6 and will be objected to if claim 6 is allowed.

II. Claims 1-9 and 11-14 are rejected under 35 U.S.C. §103(a) as unpatentable over U.S. Patent Number 5,778,248 to Leung ("Leung") in view of U.S. Patent No. 5,974,526 to Garg et al. ("Garg").

III. Claims 10, 15 and 16 is rejected under 35 U.S.C. §103(a) as unpatentable over U.S. Patent Number 5,996,063 to Gaertner et al. ("Gaertner") in view of Garg.

Argument

I. CLAIM 10 IS NOT SUBSTANTIALLY DUPLICATIVE TO CLAIM 6

The Examiner alleges that claims 6 and 10 are substantially duplicative. FOA, pg. 2.

The Appellants respectfully submit that both claims recite different subject matter. For example, claim 10 recites, in part, "a third computer readable code embodied in tangible media for assigning an entry in the temporary buffer to the logic source address of said current instruction if the no-dependency signal is not active." Claim 10 further recites, "a fourth computer readable code embodied in tangible media for issuing the instruction operand data to an instruction execution unit without assigning the entry in the temporary buffer to the logic source address of said current instruction if the

no-dependency signal is active." These claim elements are not present in claim 6.

Furthermore, claims 6 and 10 are rejected on separate grounds. Claim 6 is rejected under 35 U.S.C. §103(a) as unpatentable over U.S. Patent Number 5,778,248 in view of U.S. Patent No. 5,974,526. Claim 10, on the other hand, is rejected under 35 U.S.C. §103(a) as unpatentable over U.S. Patent Number 5,996,063 in view of U.S. Patent No. 5,974,526. It is inconsistent to argue claims 6 and 10 are substantially duplicative, yet reject the claims under different grounds.

Thus, claims 6 and 10 are not believed to be substantially duplicative.

II. CLAIMS 1-9 AND 11-14 ARE PATENTABLE OVER LEUNG IN VIEW OF GARG

Claims 1-9 and 11-14 were rejected under 35 USC §103 as being obvious over U.S. Patent No. 5,778,248 to Leung ("Leung") in view of U.S. Patent No. 5,974,526 to Garg et al. ("Garg"). FOA, pg. 3 and 8.

A *prima facie* case for obviousness can only be made if the combined reference documents teach or suggest all the claim limitations. MPEP 2143. Furthermore, to establish a *prima facie* case of obviousness, there must be some suggestion or motivation to modify the reference or to combine reference teachings. MPEP 2143.

Claim 1

Claim 1 recites, in part, "for detection of a dependency, determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction, said one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction; generating a no-dependency signal associated with said current instruction; bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active." In rejecting claim 1, the Final Office Action alleges Leung provides such a teaching. In

support of this argument, the Abstract of Leung is cited, along with column 1, lines 5-40, column 1, line 66 to column 2, line 31, column 3, lines 11-25, column 4, lines 20-49, and Figures 2-4. No further explanation is provided by the Examiner.

The Abstract of Leung states:

> A method and apparatus for determining data dependencies and enabling bypass logic in parallel. In particular, a given stage in a given execution unit will (1) compare its destination register to the destination registers of the initial stage in each execution unit, and (2) combine the result of the comparison with the propagated results of preceding stages in the given execution unit. The other stages are not checked, as this is covered by similar checking logic in the earlier stages, with the results being passed on to the subsequent stages.

The Leung Abstract provides that an execution unit compares its destination register to the destination registers of the initial stage in each execution unit. The execution unit combines the result of the comparison with the propagated results of preceding stages. As is evident in this passage, and the other cited passages, Leung does not perform register renaming and thus does not teach or suggest the claimed element of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction. Furthermore, there is no teaching of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The passage does not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active.

Column 1, lines 5-40 of Leung, cited by the Examiner, state:

> The present invention relates to RISC microprocessors which use multiple functional units which are pipelined, and in particular to the mechanisms used to bypass the writing of results to a register file in order to improve performance by using those results before they are written to the register file.
>
> One of the key advantages of reduced instruction set computing (RISC) microprocessors is the ability to process

> instructions at a very high rate. Part of this is due to the clock speed, and part of it is due to the use of multiple functional units in parallel, so that, in any one cycle, multiple functions are being performed on different instructions. In addition, pipelining is used so that each functional unit also moves its instructions through the pipeline each cycle. For example, a system with four function units used at a time and a nine-stage pipeline could have 9x4=36 instructions being processed at any one time. The instructions issued in the same cycle are referred to as an instruction group.
>
> The use of pipelining and multiple stages in multiple function units means that often data needed by one instruction in one stage of the pipeline is dependent upon the results of an instruction in another stage of the same or other execution unit. Accordingly, the data dependencies must be determined to ensure that the register file is written to with the correct results prior to an instruction accessing its input data which is provided by another instruction in another stage. High-speed microprocessors often provide bypass logic to enable the data to be obtained directly from the other stage, rather than waiting for it to be written to the destination register in the register file. However, for such bypass logic to be used, it must first be determined whether the stage whose writing is being bypassed is itself dependent upon the results of another instruction, which it is waiting for. Typically, data dependency check logic is provided at each stage to check whether the same destination register is used by any other instruction in the pipeline. The check is typically done serially, and can slow down the processing speed.

This passage merely discusses multistage RISC microprocessors and the problem of data dependency among registers. There is no discussion of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction. In addition, there is no teaching of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The passage does not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active.

Column 1, line 66 to column 2, line 31 of Leung, cited by the Examiner, state:

> The present invention provides a method and apparatus for determining data dependencies and enabling bypass logic in

> parallel. In particular, a given stage in a given execution unit will (1) compare its destination register to the destination registers of the initial stage in each execution unit, and (2) combine the result of the comparison with the propagated results of preceding stages in the given execution unit. The other stages are not checked, as this is covered by similar checking logic in the earlier stages, with the results being passed on to the subsequent stages.
>
> In a preferred embodiment, a single bit is used to indicate the result of the comparison, and is stored as a tag for the instruction associated with a given stage. Preferably, the comparison is done in advance of the instruction entering that stage, so that the results are available prior to any attempt to bypass the results of that stage, thus ensuring that no slowdown of the microprocessor operation will occur.
>
> In one embodiment, the data dependency check logic includes a number of comparators, with each comparator having a first input provided with the destination register address of the given stage, and a second input which receives the destination register address of each of the initial stages in the execution units. The outputs of the comparators are provided to an OR-gate. The OR-gate also receives an input from similar logic in the immediate preceding stage of the given execution unit. The output of the OR-gate is provided as a single bit to the tag for that instruction to enable or disable bypassing logic associated with that stage.
>
> For a further understanding of the nature and advantages of the invention, reference should be made to the following description taken in conjunction with the accompanying drawings.

This passage, like the Abstract, deals with determining data dependencies and enabling bypass logic in parallel. The execution unit compares its destination register to the destination registers of the initial stage in each execution unit. The execution unit further combines the result of the comparison with the propagated results of preceding stages. As discussed above, there is no discussion of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction. In addition, there is no teaching of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The passage does not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current

instruction if the no-dependency signal is active.

Column 3, lines 11-25 of Leung, cited by the Examiner, state:

> FIG. 2 illustrates in more detail the stages of some of the pipelines of FIG. 1. In particular, the floating point registers 50 are shown at the bottom, with the 5 floating point and graphical functional units being shown. The different stages of the pipeline are illustrated by the letters in line 64 in the middle of the diagram. At the top of FIG. 2, an outstanding instruction FIFO 66 is illustrated. As shown, the FIFO has 7 rows corresponding to 7 stages or cycles of the pipeline, with corresponding instructions being stored in 4 positions in each row, extending from 1 through 28 as shown. Each row thus stores an instruction group. The instructions corresponding to the positions in the FIFO will be found in the pipeline at the same cycle in the functional units. Instruction FIFO control logic 71 controls the operation of outstanding instruction FIFO 66.

The cited passage describes multiple stages of the microprocessor. There is no discussion of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction. In addition, there is no teaching of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The passage does not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active.

Column 4, lines 20-49 of Leung, cited by the Examiner, state:

> FIG. 4 illustrates the data dependency check logic 82 for a particular stage. A 1-bit latch 88 stores the result from the previous stages' data dependency check logic (the first stage would not have this). A series of comparators 90, 92 and 94 compare the destination register for the current stage on line 96 with the destination registers for initial stages R1, R2 and R3 of each of the execution units on lines 98, 100 and 102. The results of the comparators, along with the results of the previous stages' comparison in register 88, are provided to a NOR-gate 104. The output of the NOR-gate is provided to A 1-bit latch 106, which tags the instruction for the current stage. Note that a NOR gate and the inverting output of the latch are used to increase speed. Alternately, an OR gate or other logic could be used.

> The bit in latch 106, which tags the instruction, can be provided directly to enable logic for the bypass logic of the current stage, or can be added as a tag bit in a field associated with the instruction in outstanding instruction FIFO 66, as shown in FIG. 2. A separate input on a line 108 to each of the comparators enables the comparators only if the instructions have actually been confirmed as issued in each of stages R1, R2 and R3.
>
> When a subsequent bypass is attempted, the bypass logic will be appropriately enabled or disabled. Alternately, in another possible embodiment, the bypass logic may be usable, but the logic attempting to access the bypass logic from a different stage will check the tag in the instruction FIFO, and not use the bypass logic if it is tagged as not bypassable.

The cited passage describes data dependency check logic. Again, the Appellants submit there is no discussion of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction. In addition, there is no teaching of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The passage does not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active.

Finally, Figures 2-4, also cited by the Examiner, contain no disclosure of determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction or of one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction. The figures do not provide any teaching of generating a no-dependency signal associated with a current instruction or bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active.

Claim 1 further recites, in part, "addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target

register address."

In rejecting claim 1, the Final Office Action states that Leung does not teach addressing a mapping-table-entry with a logical source register address of current instructions to determine the mapped physical target register address. FOA, pg. 4. Nevertheless, the Final Office Action alleges that Garg contains teaching that describes addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address. OA, pg. 6. Specifically, the Office Action contends that such a teaching can be found somewhere in column 1, line 66 to column 2, line 15, column 3, lines 11-27, column 4, lines 17-40, column 6, lines 48-63, column 12, lines 30-58, and Figures 1 and 8 of Garg.

The Applicants respectfully disagree with the Examiner's interpretation of Garg and submit that Garg to does not teach or suggest the recited subject matter either alone or in combination with Leung.

Column 1, line 66 to column 2, line 15 of Garg state:

> When instructions are issued out of order and complete out of order, correspondence between register and values breaks down, and values conflict for register. The problem is severe when the goal of register allocation is to keep as many values in as few registers as possible. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator.
>
> Anti- and output dependencies are more properly called "storage conflicts" because reusing storage locations (including registers) causes instructions to interfere with one another even though conflicting instructions are otherwise independent. Storage conflicts constrain instruction issue and reduce performance. But storage conflicts, like other resource conflicts, can be reduced or eliminated by duplicating the troublesome resource. Garg, col. 1, ln. 66 - col. 2, ln. 15.

It is clear from inspection of this passage that no mention or suggestion is made of addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 3, lines 11-27 of Garg state:

> One technique for removing storage conflicts is by providing additional register that are used to reestablish the correspondence between registers and value. The additional registers are conventional allocated dynamically by hardware and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments. Garg, col. 3, ln. 11-27.

This passage deals with the broad concept of register naming and also does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 4, lines 17-40 of Garg state:

> A further technique for reducing dependencies is using register renaming with a reorder buffer which uses associative lookup. The associative lookup maps the register identifier to the reorder buffer entry as soon as the entry is allocated, and, to avoid output dependencies the lookup is prioritized so that only the value for the most recent assignment is obtained if the register is assigned more than once. A tag is obtained if the result is not yet available. There can be as many instances of a given register as there are reorder buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written from the reorder buffer to the register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register, the register file contains the only instance of the register, and this is the most recent instance.
>
> However, renaming with a reorder buffer relies on the associative lookup in the reorder buffer to map register identifiers to values. In the reorder buffer, the associative lookup is prioritized so that the reorder buffer always provides the most recent value in the register of interest (or a tag). The reorder buffer also writes values to the register file in order, so that, if the value is not in the reorder buffer, the register file must contain the most recent value. Garg, col. 4, ln. 17-40.

This passage deals with associated lookup in a reorder buffer and is silent to the teaching or suggestion of addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 6, lines 48-63 of Garg state:

> Out-of-order results for several instructions being executed at the same time are stored in a set of temporary buffers, rather that the file register designated by the instruction. If the DDC determines, for example, that a register that instruction 6's source is written to by instructions 2, 3 and 5, then the TAL will indicate that instruction 6 must wait for instruction 5 by outputting the "tag" of instruction 5 for instruction 6. The tag of instruction 5 shows the temporary buffer location where instruction 5's result is stored. It also contains a one bit signal (called a "done flag") that indicates if instruction S is finished or not. The TAL will output three tags for each instruction, because each instruction can have three source registers. If an instruction is not dependent on any previous instruction, the TAL will output the register file address of the instruction's input, rather an a temporary buffer's address. Garg, col. 6, ln. 48-63.

The Applicants respectfully submit that this passage does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 12, lines 30-58 of Garg state:

> Because it is possible that an instruction might get one of its inputs from a register that was written to by several other instructions, the present invention must choose which one is the real dependency. For example, if instructions 2 and 5 write to register 4 and instruction 7 reads register 4, then instruction 7 has two possible dependencies. In this case, it is assumed that since instruction 5 came after instruction 2 in the program, the programmer intended instruction 7 to use instruction 5's result and not instruction 2's. So, if an instruction can be dependent on several previous instructions, RRC 112 will consider it to be dependent on the highest numbered previous instruction.
>
> Once TAL 122 has determined where the real dependencies are, it must locate the inputs for each instruction. In a preferred embodiment of the present invention, the inputs can come from the actual register file or an array temporary buffers 116. RRC 112 assumes that if an instruction has no dependencies, its inputs are all in the register file. In this case, RRC 112 passes the IXS1, IXS2 and IXS/D addresses that came from IFIFO 102 to the

> register file. If an instruction has a dependency, then RRC 112 assumes that the data is in temporary buffers 116. Since RRC 112 knows which previous instruction each instruction depends on, and since each instruction always writes to the same place in temporary buffers 116, RRC 112 can determine where in temporary buffers 116 an instruction's inputs are stored. It sends these addresses to register file read ports 119 and register file 117 outputs the data from temporary buffers 116 so that the instruction can use it. Garg, col. 12, ln. 30-58.

Similarly, the Applicants submit that this passage does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

The Applicants further submit that Figures 1 and 8 of Garg do not teach or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address. Furthermore, Leung does not teach or suggest a renaming process.

In the "Response to Arguments", the Final Office Action further cites column 3, lines 11 to column 4, line 40 of Garg, arguing register renaming maps a logical register to a physical register, so the final value is mapped to the correct physical register. FOA, pg. 11. The Appellants respectfully disagree with the Examiner's interpretation of Garg and its application.

The cited passage of Garg states:

> One technique for removing storage conflicts is by providing additional register that are used to reestablish the correspondence between registers and value. The additional registers are conventional allocated dynamically by hardware and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.

Consider the following code sequence where "op" is an operation, "Rn" represents a numbered register, and ":=" represents assignment

R3b :=R3a op R5a (1)

R4b :=R3b +1 (2)

R3c :=R5a +1 (3)

R7b :=R3c op R4b (4)

Each assignment to a register creates a new "instance" of the register, denoted by an alphabetic subscript. The creation of a new instance for R3 in the third instruction avoids the anti- and output dependencies on the second and first instructions, respectively, and yet does not interfere with correctly supplying an operand to the fourth instruction. The assignment to R3 in the third instruction supersedes the assignment to R3 in the first instruction, causing R3c to become the new R3 seen by subsequent instructions until another instruction assigns a value to R3.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies and allows more instruction parallelism. Registers are still reused, but reuse is in line with the requirements of parallel execution. This is particularly helpful with out-of-order issue, because storage conflicts introduce instruction issue constants that are not really necessary to produce correct results. For example, in the preceding instruction sequence, renaming allows the third instruction to be issued immediately, whereas, without renaming, the instruction must be delayed until the first instruction is complete and the second instruction is issued.

Another technique for reducing dependencies is to associate a single bit (called a "scoreboard bit") with each register. The scoreboard bit is used to indicate that a register has a pending update. When an instruction is decoded that will write a register, the processor sets the associated scoreboard bit. The scoreboard bit is reset when the write actually occurs. Because there is only one scoreboard bit indicating whether or not there is a pending update, there can be only one such update for each register. The scoreboard stalls instruction decoding if a decoded instruction will update a register that already has a pending update (indicated by the scoreboard bit being set). This avoids output dependencies by allowing only one pending update to a register at any given time.

Register renaming, in contrast, uses multiple-bit tags to identify the various uncomputed values, some of which values may be destined for the same processor register (that is, the same program-visible register). Conventional renaming requires hardware to allocate tags from a pool of available tags that are not currently associated with any value and requires hardware to

> free the tags to the pool once the values have been computed. Furthermore, since scoreboarding allows only one pending update to a given register, the processor is not concerned about which update is the most recent. Garg, col. 3, ln. 11 - col. 4, ln. 40.

The cited passage relates to renaming an original register identifier in an instruction to identify a new register and correct value. There is no teaching of addressing a mapping-table-entry of a mapping table with a logical source register address of a current instruction thus determining the mapped physical target register address. Moreover, the Examiner does not argue such a teaching is found in the cited passage.

Finally, the Examiner asserts, "The passages cited were not meant to individually show this limitation but the combination of the passages showed this limitation." FOA, pg. 11. The Appellants respectfully submit that such an assertion does not relieve the Examiner from the burden of producing evidence of obviousness. See MPEP 2142 ("The examiner bears the initial burden of factually supporting any *prima facie* conclusion of obviousness."). As discussed above, none of the passages offered by the Examiner, either alone or in combination, teach or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Moreover, obviousness cannot be established by combining prior art to produce the claimed invention absent some teaching or suggestion supporting the combination. In re Fritch, 972 F.2d 1260, 1266, 23 USPQ2d 1780, 1783-84 (Fed. Cir. 1992). The mere fact that the prior art may be modified in the manner suggested by an Examiner not does make the modification obvious unless the prior art suggested the desirability of the modification. *Id.*

The Examiner argues, "A person of ordinary skill in the art at the time the invention was made would have recognized that Garg's table allows for tracking of values to renamed registers (Garg column 4, lines 33-40), thereby ensuring that the values are correctly mapped and

can be written to the physical registers correctly. Therefore, it would have been obvious to a person of ordinary skill in the art at the time the invention was made to incorporate the table of Garg in the device of Leung to ensure correct mapping of values to renamed registers and correct data being written into the physical registers." FOA, pg. 4.

The Office Action, however has not explained, and it not evident, why a person of ordinary skill in the art would have found it obvious to reconstruct Leung "to ensure correct mapping of values to renamed registers and correct data being written into the physical registers." As discussed above, Leung provides a method and apparatus for determining data dependencies and enabling bypass logic in parallel. In particular, a given stage in a given execution unit will compare its destination register to the destination registers of the initial stage in each execution unit, and combine the result of the comparison with the propagated results of preceding stages in the given execution unit. Leung, col. 1, ln. 66 - col. 2, ln. 6. The reference does not deal with renaming registers and would not benefit from "tracking of values to renamed registers." In this light, it is apparent that the only suggestion for combining Leung and Garg in the manner advanced by the Examiner stems from hindsight knowledge impermissibly derived from the Appellants' disclosure.

For at least these reasons, the Appellants respectfully assert that the Examiner has not established a *prima facie* case of obviousness for claim 1. The Appellants submit that the rejection of claim 1 is improper and respectfully request that the rejection of claim 1 be reversed by the honorable Board.

Claims 2-5, 11 and 13

If an independent claim is nonobvious under 35 U.S.C. 103, then any claim depending therefrom is nonobvious. In re Fine, 837 F.2d 1071, 5 USPQ2d 1596 (Fed. Cir. 1988).

Claims 2-5, 11 and 13 are dependent on and further limit claim 1. Since the rejection of claim 1 is believed improper, the rejections of claims 2-5, 11 and 13 are also believed improper for at least the same

reasons as claim 1.

Claim 6

Claim 6 recites similar limitations as claim 1, and was rejected under the same arguments as claim 1. As discussed for claim 1, the limitations recited in claim 6 are not found in Leung and Garg, either alone or in combination. Furthermore, there is no motivation to combine the teachings of Leung and Garg, as discussed above.

Claims 7-9, 12 and 14

If an independent claim is nonobvious under 35 U.S.C. 103, then any claim depending therefrom is nonobvious. In re Fine, 837 F.2d 1071, 5 USPQ2d 1596 (Fed. Cir. 1988).

Claims 7-9, 12 and 14 are dependent on and further limit claim 6. Since the rejection of claim 6 is believed improper, the rejections of claims 7-9, 12 and 14 are also believed improper for at least the same reasons as claim 6.

III. CLAIMS 10, 15 AND 16 ARE PATENTABLE OVER GAERTNER IN VIEW OF GARG

Claims 10, 15 and 16 were rejected under 35 U.S.C. §103(a) as obvious over U.S. Patent Number 5,996,063 to Gaertner et al. ("Gaertner") in view of Garg. FOA, pg. 9.

A *prima facie* case for obviousness can only be made if the combined reference documents teach or suggest all the claim limitations. MPEP 2143.

Claim 10

Claim 10 recites, in part, "a fifth computer readable code embodied in tangible media for addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address." In rejecting claim 10, the Final Office Action states that Gaertner does not teach addressing a mapping-table-entry with a logical source register address of current instructions to determine the mapped

physical target register address. FOA, pg. 10. Nevertheless, the Final Office Action alleges that Garg contains teaching that describes addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address. OA, pg. 6. Specifically, the Office Action contends that such a teaching can be found somewhere in column 1, line 66 to column 2, line 15, column 3, lines 11-27, column 4, lines 17-40, column 6, lines 48-63, column 12, lines 30-58, and Figures 1 and 8 of Garg. No further explanation is provided by the Examiner.

The Applicants respectfully disagree with the Examiner's interpretation of Garg and submit that Garg to does not teach or suggest the recited subject matter either alone or in combination with Leung.

Column 1, line 66 to column 2, line 15 of Garg state:

> When instructions are issued out of order and complete out of order, correspondence between register and values breaks down, and values conflict for register. The problem is severe when the goal of register allocation is to keep as many values in as few registers as possible. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator.
>
> Anti- and output dependencies are more properly called "storage conflicts" because reusing storage locations (including registers) causes instructions to interfere with one another even though conflicting instructions are otherwise independent. Storage conflicts constrain instruction issue and reduce performance. But storage conflicts, like other resource conflicts, can be reduced or eliminated by duplicating the troublesome resource. Garg, col. 1, ln. 66 - col. 2, ln. 15.

It is clear from inspection of this passage that no mention or suggestion is made of addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 3, lines 11-27 of Garg state:

> One technique for removing storage conflicts is by providing additional register that are used to reestablish the

> correspondence between registers and value. The additional registers are conventional allocated dynamically by hardware and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments. Garg, col. 3, ln. 11-27.

This passage deals with the broad concept of register naming and also does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 4, lines 17-40 of Garg state:

> A further technique for reducing dependencies is using register renaming with a reorder buffer which uses associative lookup. The associative lookup maps the register identifier to the reorder buffer entry as soon as the entry is allocated, and, to avoid output dependencies the lookup is prioritized so that only the value for the most recent assignment is obtained if the register is assigned more than once. A tag is obtained if the result is not yet available. There can be as many instances of a given register as there are reorder buffer entries, so there are no storage conflicts between instructions. The values for the different instances are written from the reorder buffer to the register file in sequential order. When the value for the final instance is written to the register file, the reorder buffer no longer maps the register, the register file contains the only instance of the register, and this is the most recent instance.
>
> However, renaming with a reorder buffer relies on the associative lookup in the reorder buffer to map register identifiers to values. In the reorder buffer, the associative lookup is prioritized so that the reorder buffer always provides the most recent value in the register of interest (or a tag). The reorder buffer also writes values to the register file in order, so that, if the value is not in the reorder buffer, the register file must contain the most recent value. Garg, col. 4, ln. 17-40.

This passage deals with associated lookup in a reorder buffer and is silent to the teaching or suggestion of addressing a mapping-table-entry of a mapping table with a logical source register address of said

current instruction thus determining the mapped physical target register address.

Column 6, lines 48-63 of Garg state:

> Out-of-order results for several instructions being executed at the same time are stored in a set of temporary buffers, rather that the file register designated by the instruction. If the DDC determines, for example, that a register that instruction 6's source is written to by instructions 2, 3 and 5, then the TAL will indicate that instruction 6 must wait for instruction 5 by outputting the "tag" of instruction 5 for instruction 6. The tag of instruction 5 shows the temporary buffer location where instruction 5's result is stored. It also contains a one bit signal (called a "done flag") that indicates if instruction S is finished or not. The TAL will output three tags for each instruction, because each instruction can have three source registers. If an instruction is not dependent on any previous instruction, the TAL will output the register file address of the instruction's input, rather an a temporary buffer's address. Garg, col. 6, ln. 48-63.

The Applicants respectfully submit that this passage does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Column 12, lines 30-58 of Garg state,

> Because it is possible that an instruction might get one of its inputs from a register that was written to by several other instructions, the present invention must choose which one is the real dependency. For example, if instructions 2 and 5 write to register 4 and instruction 7 reads register 4, then instruction 7 has two possible dependencies. In this case, it is assumed that since instruction 5 came after instruction 2 in the program, the programmer intended instruction 7 to use instruction 5's result and not instruction 2's. So, if an instruction can be dependent on several previous instructions, RRC 112 will consider it to be dependent on the highest numbered previous instruction.
>
> Once TAL 122 has determined where the real dependencies are, it must locate the inputs for each instruction. In a preferred embodiment of the present invention, the inputs can come from the actual register file or an array temporary buffers 116. RRC 112 assumes that if an instruction has no dependencies, its inputs are all in the register file. In this case, RRC 112 passes the IXS1, IXS2 and IXS/D addresses that came from IFIFO 102 to the register file. If an instruction has a dependency, then RRC 112 assumes that the data is in temporary buffers 116. Since RRC 112 knows which previous instruction each instruction depends on, and since each instruction always writes to the same place in

> temporary buffers 116, RRC 112 can determine where in temporary buffers 116 an instruction's inputs are stored. It sends these addresses to register file read ports 119 and register file 117 outputs the data from temporary buffers 116 so that the instruction can use it. Garg, col. 12, ln. 30-58.

Similarly, the Applicants submit that this passage does not mention or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

The Applicants further submit that Figures 1 and 8 of Garg do not teach or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address. Furthermore, Leung does not teach or suggest a renaming process.

In the "Response to Arguments", the Final Office Action further cites column 3, lines 11 to column 4, line 40 of Garg, arguing register renaming maps a logical register to a physical register, so the final value is mapped to the correct physical register. FOA, pg. 11. The Appellants respectfully disagree with the Examiner's interpretation of Garg and its application.

The cited passage of Garg states:

> One technique for removing storage conflicts is by providing additional register that are used to reestablish the correspondence between registers and value. The additional registers are conventional allocated dynamically by hardware and the registers are associated with values needed by the program using "register renaming." To implement register renaming, processors typically allocate a new register for every new value produced (i.e., for every instruction that writes a register). An instruction identifying the original register, for the purpose of reading its value, obtains instead the value in the newly allocated register. Thus, hardware renames the original register identifier in the instruction to identify the new register and correct value. The same register identifier in several different instructions may access different hardware registers, depending on the locations of register references with respect to register assignments.
>
> Consider the following code sequence where "op" is an operation, "Rn" represents a numbered register, and ":=" represents assignment

R3b :=R3a op R5a (1)

R4b :=R3b +1 (2)

R3c :=R5a +1 (3)

R7b :=R3c op R4b (4)

Each assignment to a register creates a new "instance" of the register, denoted by an alphabetic subscript. The creation of a new instance for R3 in the third instruction avoids the anti- and output dependencies on the second and first instructions, respectively, and yet does not interfere with correctly supplying an operand to the fourth instruction. The assignment to R3 in the third instruction supersedes the assignment to R3 in the first instruction, causing R3c to become the new R3 seen by subsequent instructions until another instruction assigns a value to R3.

Hardware that performs renaming creates each new register instance and destroys the instance when its value is superseded and there are no outstanding references to the value. This removes anti- and output dependencies and allows more instruction parallelism. Registers are still reused, but reuse is in line with the requirements of parallel execution. This is particularly helpful with out-of-order issue, because storage conflicts introduce instruction issue constants that are not really necessary to produce correct results. For example, in the preceding instruction sequence, renaming allows the third instruction to be issued immediately, whereas, without renaming, the instruction must be delayed until the first instruction is complete and the second instruction is issued.

Another technique for reducing dependencies is to associate a single bit (called a "scoreboard bit") with each register. The scoreboard bit is used to indicate that a register has a pending update. When an instruction is decoded that will write a register, the processor sets the associated scoreboard bit. The scoreboard bit is reset when the write actually occurs. Because there is only one scoreboard bit indicating whether or not there is a pending update, there can be only one such update for each register. The scoreboard stalls instruction decoding if a decoded instruction will update a register that already has a pending update (indicated by the scoreboard bit being set). This avoids output dependencies by allowing only one pending update to a register at any given time.

Register renaming, in contrast, uses multiple-bit tags to identify the various uncomputed values, some of which values may be destined for the same processor register (that is, the same program-visible register). Conventional renaming requires hardware to allocate tags from a pool of available tags that are not currently associated with any value and requires hardware to free the tags to the pool once the values have been computed. Furthermore, since scoreboarding allows only one pending update to a given register, the processor is not concerned about which update is the most recent. Garg, col. 3, ln. 11 - col. 4, ln.

40.

The cited passage relates to renaming an original register identifier in an instruction to identify a new register and correct value. There is no teaching of addressing a mapping-table-entry of a mapping table with a logical source register address of a current instruction thus determining the mapped physical target register address. Moreover, the Examiner does not argue such a teaching is found in the cited passage.

Finally, the Examiner asserts, "The passages cited were not meant to individually show this limitation but the combination of the passages showed this limitation." FOA, pg. 11. The Appellants respectfully submit that such an assertion does not relieve the Examiner from the burden of producing evidence of obviousness. See MPEP 2142 ("The examiner bears the initial burden of factually supporting any *prima facie* conclusion of obviousness."). As discussed above, none of the passages offered by the Examiner, either alone or in combination, teach or suggest addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Moreover, obviousness cannot be established by combining prior art to produce the claimed invention absent some teaching or suggestion supporting the combination. In re Fritch, 972 F.2d 1260, 1266, 23 USPQ2d 1780, 1783-84 (Fed. Cir. 1992).

The Examiner argues, "Therefore, it would have been obvious to a person of ordinary skill in the art at the time the invention was made to incorporate the table of Garg in the device of Leung to ensure correct mapping of values to renamed registers and correct data being written into the physical registers." FOA, pg. 10 (emphasis added). The Appellants submit the Examiner has not provided any reasoning why a person of ordinary skill in the art would have found it obvious to reconstruct Gaertner with the teachings of Garg.

For at least these reasons, the Appellants respectfully assert that the Examiner has not established a *prima facie* case of obviousness

for claim 1. The Appellants submit that the rejection of claim 1 is improper and respectfully request that the rejection of claim 1 be reversed by the honorable Board.

Claims 15 and 16

If an independent claim is nonobvious under 35 U.S.C. 103, then any claim depending therefrom is nonobvious. In re Fine, 837 F.2d 1071, 5 USPQ2d 1596 (Fed. Cir. 1988).

Claims 15 and 16 are dependent on and further limit claim 10. Since the rejection of claim 10 is believed improper, the rejections of claims 15 and 16 are also believed improper for at least the same reasons as claim 10.

Conclusion

In view of the foregoing, Appellant submits that the rejections of claims 1-16 are improper and respectfully requests that the rejections of claims 1-16 be reversed by the Board.

Respectfully submitted,

/ido tuchman/

Dated: July 9, 2007

Ido Tuchman, Reg. No. 45,924
Law Office of Ido Tuchman
82-70 Beverly Road
Kew Gardens, NY 11415
Telephone (718) 544-1110
Facsimile (866) 607-8538

Claims Appendix

Claim 1. (previously presented) A method for operating an out-of-order processor comprised of an instruction pipeline, the method comprising the steps of:

for detection of a dependency, determining for each current instruction involved in a renaming process that a logic target address of one or more instructions is not the same as a logic source address of said current instruction, said one or more instructions being stored in a temporary buffer associated with a pipeline process downstream of the current instruction;

generating a no-dependency signal associated with said current instruction;

bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active; and

addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Claim 2. (previously presented) The method according to claim 1 in which the step of generating a no-dependency signal comprises the steps of:

comparing a plurality of logic target register addresses and the logic source register address of the current instruction;

in case the logic target register addresses and the logic source register address match, setting the no-dependency signal to not active; and

generating a dependency signal for the respective source register.

Claim 3. (previously presented) The method according to claim 1 further comprising the step of evaluating 'valid'-bits of speculative target registers stored in a storage associated with speculatively calculated instruction result data to generate the no-dependency signal.

Claim 4. (previously presented) The method according to claim 1 further comprising:

reading a committed-status flag in said entry;

comparing the logic target register address and the logic source register address of the current instruction in case the no-dependency signal is not active; and

generating a dependency signal for the respective source register.

Claim 5. (previously presented) The method according to claim 1

further comprising:

reading a committed-status flag in said entry;

comparing the logic target register address and the logic source register address of the current instruction;

in case the logic target register address and the logic source register address match, setting the no-dependency signal to not active; and

generating a dependency-signal for the respective source register.

Claim 6. (previously presented) A processing system having means for executing a readable machine language, said readable machine language comprises:

a first computer readable code embodied in tangible media for the detection of a dependency, determining for each current instruction involved in a renaming process that a logic target address of one or more instructions stored in a temporary buffer associated with a pipeline process downstream of the current instruction is not the same as a logic source address of said current instruction,

a second computer readable code embodied in tangible media for generating a no-dependency signal associated with said current instruction,

a third computer readable code embodied in tangible media for bypassing a portion of the instruction pipeline for the current instruction if the no-dependency signal is active, and

a fourth computer readable code embodied in tangible media for addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Claim 7. (previously presented) The processing system according to claim 6 in which in case of a content-addressable memory (CAM)-based renaming scheme the first computer readable code for determining the dependency of a current instruction comprises a compare logic in which all instructions to be checked for dependency are involved and an OR gate coupled with the compare logic.

Claim 8. (previously presented) The processing system according to claim 7 further comprising a plurality of AND gates the input of which comprises a target register ‘valid bits’ signal and a respective compare logic output signal.

Claim 9. (previously presented) The processing system according to

claim 6 in which the case of a mapping-table-based renaming scheme each mapping table entry comprises an additional instruction-commited flag, and the first computer readable code for determining the dependency of a current instruction comprises a logic for ANDing a target register 'valid bits' signal in which all instructions to be checked for dependency are involved and an OR gate coupled with the logic.

Claim 10. (previously presented) A computer system having an out-of-order processing system, said computer system executes a readable machine language, said readable machine language comprises:

a first computer readable code embodied in tangible media for the detection of a dependency, determining for each current instruction involved in a renaming process that a logic target address of one or more instructions stored in a temporary buffer associated with a pipeline process downstream of the current instruction is not the same as a logic source address of said current instruction,

a second computer readable code embodied in tangible media for generating a no-dependency signal associated with said current instruction,

a third computer readable code embodied in tangible media for assigning an entry in the temporary buffer to the logic source address of said current instruction if the no-dependency signal is not active;

a fourth computer readable code embodied in tangible media for issuing the instruction operand data to an instruction execution unit without assigning the entry in the temporary buffer to the logic source address of said current instruction if the no-dependency signal is active; and

a fifth computer readable code embodied in tangible media for addressing a mapping-table-entry of a mapping table with a logical source register address of said current instruction thus determining the mapped physical target register address.

Claim 11. (previously presented) The method according to claim 1, further comprising partitioning the current instruction into a reorder buffer and an architectural register array, the reorder buffer configured to store speculative results of the current instruction and the architectural register array configured to store an architectural state of the processor.

Claim 12. (previously presented) The method according to claim 11, wherein the no-dependency signal is logically ANDed with a plurality of validity bits indicating data is available at the reorder buffer and the

architectural register array.

Claim 13. (previously presented) The processing system according to claim 6, further comprising a fifth computer readable code embodied in tangible media for partitioning the current instruction into a reorder buffer and an architectural register array, the reorder buffer configured to store speculative results of the current instruction and the architectural register array configured to store an architectural state of the processor.

Claim 14. (previously presented) The processing system according to claim 13, wherein the no-dependency signal is logically ANDed with a plurality of validity bits indicating data is available at the reorder buffer and the architectural register array.

Claim 15. (previously presented) The computer system according to claim 10, further comprising a sixth computer readable code embodied in tangible media for partitioning the current instruction into a reorder buffer and an architectural register array, the reorder buffer configured to store speculative results of the current instruction and the architectural register array configured to store an architectural state of the processor.

Claim 16. (previously presented) The computer system according to claim 15, wherein the no-dependency signal is logically ANDed with a plurality of validity bits indicating data is available at the reorder buffer and the architectural register array.

Evidence Appendix

None.

Related Proceedings Appendix

None.